PRACE

Profiling techniques for parallel applications

Analyzing program performance with HPCToolkit

Introduction

- Thomas Ponweiser
- Johannes Kepler University Linz (JKU)
- Involved in PRACE 3IP, WP 7
  - Subtask “Debugging and Profiling techniques”
  - Support expert for “Preparatory Access Type C”
- Personal background:
  - Software developer since 2008
  - Currently finishing my study of Technical Mathematics

Introduction

- Focus of this session
  - Profiling of parallel applications
    - Statistical sampling
    - Introduction to HPCToolkit
  - Strategies for finding optimization potential (not limited to HPCToolkit)
    - "High penalty" and "Waste" metrics
    - "Profiling using expectations"

Outline

- Overview: Basic profiling techniques
  - Statistical sampling vs. Code instrumentation
- HPCToolkit: A quick introduction
- Effective analysis strategies
  - Pinpointing inefficiencies
  - Pinpointing scalability bottlenecks
- Practical part
  - Analysis of program profiles (`hpcviewer`)
  - Analysis of program traces (`hpctraceviewer`)

Prerequisites for Practical Part

- Download HPCToolkit profile and trace viewers
  - http://hpctoolkit.org/software.html
  - `hpcviewer-5.3.2`
  - `hpctraceviewer-5.3.2`
  - Try to launch them (Java required)
- Download prepared profiles

Outline

- Overview: Basic profiling techniques
  - Statistical sampling vs. Code instrumentation
- HPCToolkit: A quick introduction
- Effective analysis strategies
  - Pinpointing inefficiencies
  - Pinpointing scalability bottlenecks
- Practical part
  - Analysis of program profiles (`hpcviewer`)
  - Analysis of program traces (`hpctraceviewer`)

Overview

Statistical sampling

- Sampling:
  - Program flow is periodically interrupted, current program state is examined.
- Asynchronous sampling:
  - Timers
  - Hardware counters
    (CPU cycles, L3 cache misses, etc.)
- Synchronous sampling:
  - Calls to certain library functions are intercepted (`malloc`, `fread`, …)

Code Instrumentation

- Instrumentation:
  - Code for collecting profiling information is inserted into the original program.
- Approaches:
  - Manual (measurement APIs)
  - Automatic source level
  - Compiler assisted (e.g. gprof)
  - Binary translation
  - Runtime instrumentation

Overview

Statistical sampling

- Sampling:
  - Program flow is periodically interrupted, current program state is examined.
- Asynchronous sampling:
  - Timers
  - Hardware counters
    (CPU cycles, L3 cache misses, etc.)
- Synchronous sampling:
  - Calls to certain library functions are intercepted (`malloc`, `fread`, …)

Code Instrumentation

- Instrumentation:
  - Code for collecting profiling information is inserted into the original program.
- Approaches:
  - Manual (measurement APIs)
  - Automatic source level
  - Compiler assisted (e.g. gprof)
  - Binary translation
  - Runtime instrumentation

Statistical sampling: Advantages

- No changes to program or build process
  - Recommended: Debugging symbols
- No blind spots: Measurements cover
  - Library functions
  - Functions with unavailable source code
- Low overhead
  - typically 3 to 5%

Statistical sampling: Limitations

- Statistical sampling involves some degree of uncertainty
  - Information attributed to source lines may not be accurate
- Certain types of information not available:
  - Number of calls of a certain function
  - Average runtime per call of a certain function

Outline

- Overview: Basic profiling techniques
  - Statistical sampling vs. Code instrumentation
- HPCToolkit: A quick introduction
- Effective analysis strategies
  - Pinpointing inefficiencies
  - Pinpointing scalability bottlenecks
- Practical part
  - Analysis of program profiles (hpcviewer)
  - Analysis of program traces (hpctraceviewer)

HPCToolkit: A quick introduction

- Suite of tools for program performance analysis
- Developed at Rice University, Houston, Texas
- Features
  - Statistical sampling
  - Full call-path unwinding
  - Attribution of metrics at the level of functions, loops and source lines
  - Computation of user-defined metrics

HPCToolkit: A quick introduction

- Supports
  - Asynchronous sampling
    - System timers, Hardware counters (PAPI library)
  - Synchronous sampling (via `LD_PRELOAD`)

- Suited for
  - Threaded applications
  - MPI applications
  - Hybrid applications (Threading + MPI)

HPCToolkit: Basic workflow

Step	Command	Description
(1)	`hpcrun`	Measures program performance
(2)	`hpcstruct`	Recovers program structure from the binary
(3)	`hpcprof` / `hpcprof-mpi`	Creates an experiment database
(4)	`hpcviewer` / `hpctraceviewer`	Displays experiment database (profile or trace view)

Step (1) – Performance measurement

```
# A) Sequential or threaded applications:
hpcrun [options] command [args]

# B) MPI or hybrid applications:
mpirun [mpi-opts] hpcrun [options] command [args]

# Important options:
# -e event@period  ... Specify sampling sources
# -t ................. Enable trace data collection
# -f frac ............ Enable measurement only with probability frac.
#                      Supported number formats: 0.1 or 1/10
# -o outpath ......... Specify measurement output directory

# Example - sample every ~4 million cpu cycles:
mpirun -n 4 hpcrun -e PAPI_TOT_CYC@4100100 ./myprog --some-arg
```

PARTNERSHIP FOR ADVANCED COMPUTING IN EUROPE

Step (2): Program structure recovery

```
# Analyze program structure (recovers loops from optimized binaries):
hpcstruct [options] binary

# Example:
hpcstruct ./myprog
```

Step (3): Experiment database creation

```
# Join (i) measurements, (ii) program structure and (iii) source code
# together in a so-called "experiment database"

# Three alternatives:
# (a) threaded or small MPI executions
hpcprof [options] measurement-directory...

# (b) medium size MPI executions
hpcprof-mpi [options] measurement-directory...

# (c) large MPI executions
mpirun [mpi-opts] hpcprof-mpi [options] measurment-directory...
```

Step (3): Experiment database creation

```
# Important options for hpcprof and hpcprof-mpi:

# -I path-to-source ... Location of source code
# -S structure-file ... Specify the file generated by hpcstruct
# -o outpath .......... Name of the experiment database directory
# -M metric ........... Aggregation level for metric output:
#    sum .................. Only metric sums
#    stats ................ Sum, mean, stddev, min, max for each metric
#    thread ............... Per-thread/process info (no aggregation)

# Example:
hpcprof -I ./src/'*' -S myprog.hpcstruct -M stats measurments
```

Step (3): Experiment database creation

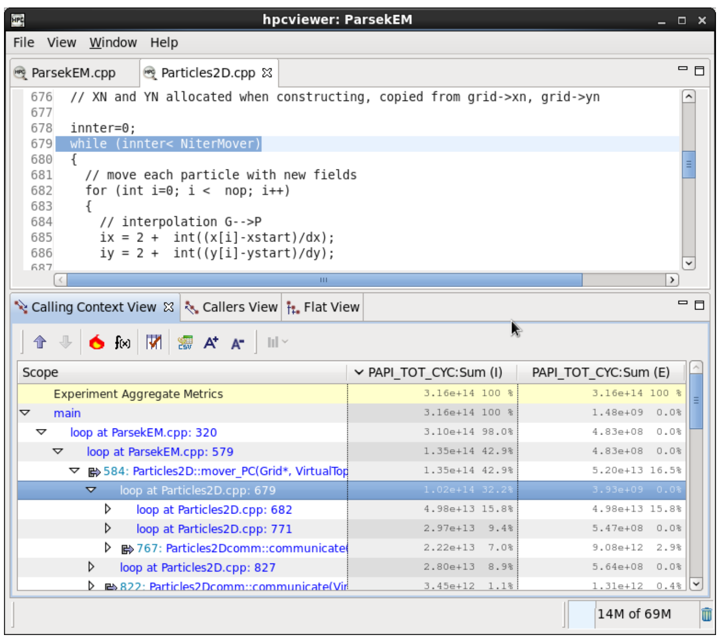
`hpcprof` vs. `hpcprof-mpi`

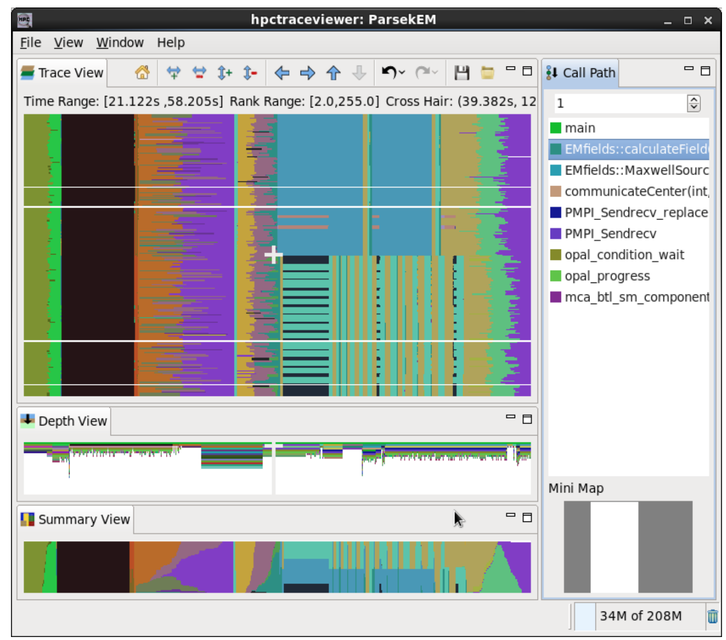
- Option `–M thread`
  - Not supported by `hpcprof-mpi`
- Per-Process/Thread metric creation
  - Only supported by `hpcprof-mpi`
  - Enables metric plots and histograms in profile viewer
  - Profiles generated with `hpcprof-mpi` are larger

Step (4): Profile analysis

```
# Profile analysis
hpcviewer experiment-database
```

```
# Trace analysis
hpctraceviewer experiment-database
```





HPCToolkit: An example

```
# (1) Measure performance of ./myprog running with 4 and 8 MPI processes
mpirun -n 4 hpcrun -o m4 -e PAPI_TOT_CYC@4100100 ./myprog --some-arg
mpirun -n 8 hpcrun -o m8 -e PAPI_TOT_CYC@4100100 ./myprog --some-arg

# (2) Program structure recovery; generates ./myprog.hpcstruct
hpcstruct ./myprog

# (3) Metric attribution
hpcprof -S myprog.hpcstruct –I ./src/'*' -o db-4-8 m4 m8

# (4) View profile
hpcviewer db-4-8
```

Outline

- Overview: Basic profiling techniques
  - Statistical sampling vs. Code instrumentation
- HPCToolkit: A quick introduction
- Effective analysis strategies
  - Pinpointing inefficiencies
  - Pinpointing scalability bottlenecks
- Practical part
  - Analysis of program profiles (`hpcviewer`)
  - Analysis of program traces (`hpctraceviewer`)

Selecting sampling sources

- Questions:
  1. Which sampling sources are available?
  2. Which sampling source(s) should I select?
  3. What is an appropriate sampling frequency?

Selecting sampling sources

- Questions:
  1. Which sampling sources are available?
  2. Which sampling source(s) should I select?
  3. What is an appropriate sampling frequency?

(1) Available sampling sources

```
# List available sampling sources:
hpcrun -l

# Output (shortened):

===========================================================================
Available Timer events
===========================================================================
Name                Description
---------------------------------------------------------------------------
WALLCLOCK           Wall clock time used by the process in microseconds.
REALTIME            Real clock time used by the thread in microseconds.
CPUTIME             CPU clock time used by the thread in microseconds.

Note: do not use multiple timer events in the same run.
```

(1) Available sampling sources

```
==========================================================================
Available PAPI preset events
==========================================================================
Name        Profilable  Description
--------------------------------------------------------------------------
PAPI_TOT_CYC    Yes     Total cycles
PAPI_STL_ICY    Yes     Cycles with no instruction issue
...
PAPI_L3_TCM     Yes     Level 3 cache misses
...
PAPI_BR_CN      Yes     Conditional branch instructions
PAPI_BR_MSP     Yes     Conditional branch instructions mispredicted
...
PAPI_FP_INS     No      Floating point instructions
PAPI_FDV_INS    Yes     Floating point divide instructions
...
```

(1) Available sampling sources

```
==========================================================================
Other available events
==========================================================================
Name                Description
--------------------------------------------------------------------------
RETCNT              Each time a procedure returns, the return count for that
                    procedure is incremented
                    (experimental feature, x86 only)

MEMLEAK             The number of bytes allocated and freed per dynamic context

IO                  The number of bytes read and written per dynamic context
```

Selecting sampling sources

- Questions:
  1. Which sampling sources are available?
  2. Which sampling source(s) should I select?
  3. What is an appropriate sampling frequency?

(2) Selecting sampling sources

Most important sampling source:	
PAPI_TOT_CYC	CPU cycles (Measures execution time) Alternatives: • WALLCLOCK • REALTIME • CPUTIME

- My experience:
  - Most problems are traceable just by looking at execution time (`PAPI_TOT_CYC`).

(2) Selecting sampling sources

Sampling sources for detecting inefficiencies:	
PAPI_STL_ICY	CPU cycles without activity (waiting times)
PAPI_L3_TCM	L3 Cache misses (inefficient data access patterns) Solutions: Data restructuring, Loop tiling, …
PAPI_FP_INS, PAPI_FDV_INS, …	Floating point instructions
IO	Bytes read/written
PAPI_BR_CN, PAPI_BR_MSP	Branch misprediction

(2) Selecting sampling sources

Other potentially interesting sampling sources:	
MEMLEAK	Allocated/freed bytes, may be used for debugging
RETCNT	Number of times a function is being called

- My experience:
  - `MEMLEAK` can be helpful for debugging, but does not always work.
    - Had problems when running with OpenMPI.

Selecting sampling sources

- Questions:
  1. Which sampling sources are available?
  2. Which sampling source(s) should I select?
  3. What is an appropriate sampling frequency?

(3) Selecting the sampling frequency

- Rules of thumb:
  - Between 10 and 1000 samples per second and process (or thread).
  - More than 1000 samples/s
    - can distort the profiling results
    - make profiles/traces unnecessary big
  - Profiling overhead should remain below 5%.
  - For profiling:
    - Longer runs with lower frequency
  - For tracing:
    - Shorter runs with higher frequency

(3) Selecting the sampling frequency

- Formula for PAPI_TOT_CYC:
  - [CPU GHz] $\times 10^4$ … 10 samples / s
  - [CPU GHz] $\times 10^6$ … 1000 samples / s
  - Choose something in between
- Good frequencies for other metrics are always application and problem dependent
- For synchronous events (IO, MEMLEAK) no sampling frequency needs to be specified

Performance analysis strategies

- Detecting inefficiencies:
  - Monitor “high-penalty” events, e.g.
    - `PAPI_L3_TCM`
    - `PAPI_STL_ICY`
  - Define your own “waste metrics”
    - E.g. “Missed floating point opportunities”:
    - `2 × PAPI_TOT_CYC – PAPI_FP_INS`

Performance analysis strategies

- Detecting scalability bottlenecks:
  "Profiling using expectations"
  - Define your own metrics, reflecting your expectations

- Example: Strong scaling
  - Experiment database with measurements for N and 2N processes (fixed problem size)
  - Define your own metric for parallel overhead, e.g.
    - `OVERHEAD = PAPI_TOT_CYC(2N) - PAPI_TOT_CYC(N)`

Performance analysis strategies

- Further reading:
  - HPCToolkit User's Manual
    - http://hpctoolkit.org/documentation.html
  - References given in User's Manual
    - In particular [3], [5], [8], [9].

Outline

- Overview: Basic profiling techniques
  - Statistical sampling vs. Code instrumentation
- HPCToolkit: A quick introduction
- Effective analysis strategies
  - Pinpointing inefficiencies
  - Pinpointing scalability bottlenecks
- Practical part
  - Analysis of program profiles (`hpcviewer`)
  - Analysis of program traces (`hpctraceviewer`)

Detecting inefficiencies (1/4)

- Go to directory `1-inefficiency`
- Open `1a-before-simple` with `hpcviewer`.
  - What is the “hot path” w.r.t. execution time?
  - Within the routine `mover_PC`, which lines of code are long-running?
  - Do you spot optimization potential?
- Close experiment database.

Detecting inefficiencies (2/4)

- Stay in directory `1-inefficiency`
- Open `1a-before-allmetrics` with `hpcviewer`.
  - Deselect exclusive metric columns for display
  - What is the "hot path" with respect to
    - Stalled CPU Cycles?
    - L3 Cache misses?
- Leave database open.

Detecting inefficiencies (3/4)

- In opened database, `1a-before-allmetrics`
  - Deselect all columns except
    - PAPI_TOT_CYC:Sum (I)
    - PAPI_FP_INS:Sum (I)
  - Define a metric for missed floating point opportunities
    - `FPWASTE = 2 × PAPI_TOT_CYC - PAPI_FP_INS`
  - What is the "hot path" w.r.t. `FPWASTE`?
- Leave database open.

Detecting inefficiencies (4/4)

- In addition to `1a-before-allmetrics`, open database `1b-after-allmetrics`.
  - Do the same for 1b-after-allmetrics as for 1a-before-allmetrics:
    - Display only PAPI_TOT_CYC:Sum (I) and PAPI_FP_INS:Sum (I)
    - Define metric `FPWASTE`
- Compare databases: Execution time and `FPWASTE`
  - Of whole run (main)
  - Of function `mover_PC`
  - What has changed in the source code of `mover_PC`?
- Close both databases.

Detecting load imbalance (1/1)

- Go to directory `2-imbalance`.
- Open `trace-totcyc-stats` with `hpcviewer`.
  - Display only PAPI_TOT_CYC:Mean (I) and PAPI_TOT_CYC:Max (I).
  - Define metric `IMBALANCE`:
    - `PAPI_TOT_CYC:Max (I) / PAPI_TOT_CYC:Mean (I)`
- Within the longest-running loop of `main`:
  - Do you spot a routine with high runtime and high `IMBALANCE`?
- Close database, and re-open with `hpctraceviewer`.
  - Do you find the routine in the trace?
  - What is happening?

Pinpointing scalability bottlenecks (1/2)

- Go to directory `3-scalbility`
- Open `1-before-128-256` with `hpcviewer`
  - Define a metric `OVERHEAD` as the difference of:
    - `2.PAPI_TOT_CYC:Sum (I)` (256 procs)
    - `1.PAPI_TOT_CYC:Sum (I)` (128 procs)
  - What are the “hot paths” w.r.t. execution time and `OVERHEAD`?
- Leave database open.

Pinpointing scalability bottlenecks (1/2)

- In addition to `1-before-128-256`, open `2-after-128-256`
  - How has the overall runtime changed?
  - Has the hot path w.r.t. execution time changed?
  - How has the source code changed in `exchange.c`?
- Close both databases.

Debugging

- Go to directory 4-debugging
- Open `profile-mem-io` with `hpcviewer`.
  - Which routines read/write most of the data?
  - Plot different metrics for `main`.
- Close database.

References

- HPCToolkit documentation:
  - http://hpctoolkit.org/documentation.html